LU05e - Klassenbasierte Decorators in Python

Klassenbasierte Decorators sind eine leistungsfähige Ergänzung zu den herkömmlichen, funktionsbasierten Decorators in Python. Sie bieten mehr Flexibilität und können leichter wiederverwendet und erweitert werden. In dieser Theorieseite werden wir die Grundlagen und Anwendungsbeispiele für klassenbasierte Decorators besprechen.

Grundkonzepte

Ein klassenbasierter Decorator ist eine Klasse, die eine Methode namens `__call__` implementiert. Diese Methode wird aufgerufen, wenn der Decorator auf eine Funktion angewendet wird. Optional kann die Klasse auch eine `__init__`-Methode implementieren, um die zu dekorierende Funktion als Argument zu erhalten.

Syntax

Die grundlegende Syntax für einen klassenbasierten Decorator in Python ist die folgende:

```
class MyDecorator:
    def __init__(self, original_function):
        self.original_function = original_function

    def __call__(self, *args, **kwargs):
        print(f"Vor dem Aufruf von {self.original_function.__name__}")
        result = self.original_function(*args, **kwargs)
        print(f"Nach dem Aufruf von {self.original_function.__name__}")
        return result

@MyDecorator
def display():
    print("Die Display-Funktion wurde aufgerufen.")
```

In diesem Beispiel wird der Decorator `MyDecorator` auf die Funktion `display` angewendet. Der „@“-Operator ist hier syntaktischer Zucker für `display = MyDecorator(display)`.

Anwendungsbeispiele

Logging Decorator

Ein einfaches Beispiel für einen klassenbasierten Decorator könnte ein Logging-Decorator sein, der Informationen vor und nach dem Aufruf einer Funktion ausgibt.

```
class LoggingDecorator:
    def __init__(self, original_function):
        self.original_function = original_function

    def __call__(self, *args, **kwargs):
        print(f"Logging vor dem Aufruf von
{self.original_function.__name__}")
        result = self.original_function(*args, **kwargs)
        print(f"Logging nach dem Aufruf von
{self.original_function.__name__}")
        return result

@LoggingDecorator
def add(a, b):
    return a + b
```

In diesem Beispiel gibt der `LoggingDecorator` Meldungen aus, bevor und nachdem die `add`-Funktion aufgerufen wird.

Timing Decorator

Ein weiteres Beispiel könnte ein Timing-Decorator sein, der die Ausführungszeit einer Funktion misst.

```
import time

class TimingDecorator:
    def __init__(self, original_function):
        self.original_function = original_function

    def __call__(self, *args, **kwargs):
        start_time = time.time()
        result = self.original_function(*args, **kwargs)
        end_time = time.time()
        print(f"{self.original_function.__name__} ran in {end_time -
start_time} seconds")
        return result

@TimingDecorator
def calculate_factorial(num):
    factorial = 1
    for i in range(1, num + 1):
        factorial *= i
    return factorial
```

Hier misst der `TimingDecorator` die Ausführungszeit der `calculate_factorial`-Funktion und gibt sie aus.

Diese Theorieseite sollte einen umfassenden Überblick über das Konzept der klassenbasierten Decorators in Python bieten. Mit diesen Decorators können Sie komplexe und wiederverwendbare

Funktionalitäten in Ihren Python-Anwendungen implementieren.

M323-LU05